12-04-00

A

Certificate of Mailing By "U.S. Express Mail" Under 37 C.F.R. 1.10(c)

"EXPRESS MAIL" Mailing Label Number: EL547553748US Date of Deposit. 11/28/2000

I hereby certify that this paper and/or fee is being deposited with the United States Postal Service "EXPRESS MAIL POST OFFICE TO ADDRESSEE" service under 37 C.F.R. 1.10 on the date indicated above and is addressed to the Assistant Commissioner For Patents, Washington, DC 20231.

Name: Maritza Kidd

Signature 11/28/2000 Signature Date

Docket No. : SIA-P032

APPLICATION TRANSMITTAL LETTER

Assistant Commissioner of Patents
United States Patent and Trademark Office
Washington, D.C. 20231
ATTN: BOX PATENT APPLICATION

Sir:

Transmitted herewith for filing is the patent application of
Inventor(s): **NADJ, et al.**
Entitled: **DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES**

_31_ No. pages of specification, including title page, claims and abstract
_13_ No. sheets of _X_ informal, _____formal drawings

Also enclosed are:

_X_ Unexecuted Combined Declaration and Power of Attorney for Patent Application
_X_ An unexecuted Assignment of the Application
___ Form PTO-1595 (Recordation Cover Sheet for Assignment)
_X_ Verified Statement Claiming Small Entity Status with Cover Sheet
___ An Information Disclosure Statement (Form PTO-1449A and Form PTO-1449B)
___ A copy of References cited in Information Disclosure: _____ documents

FEES DUE

The fees due for filing the application pursuant to 37 C.F.R. 1.16 and for recording the Assignment, if any, are determined as follow:

CLAIMS					
	No. of Claims		Extra Claims	Rate	Fees
Basic Application Fee ($710 large entity; $355 small entity)					$ 355.00
Total Claims	12	Minus 20 =		X $18 = X $ 9 (small) =	0.00
Total Independent Claims	1	Minus 3 =		X $78 = X $39 (small) =	0.00
If Multiple Dependent Claims are presented, add $260.00 or $130.00(small)					
If Assignment enclosed, add Assignment Recording Fee $40.00					40.00
TOTAL APPLICATION FEE DUE					$ 395.00

PAYMENT OF FEES

The full fee due in connection with this communication is $ 395.00
and is provided as follows:

____ The Commissioner is hereby authorized to charge the fees associated with this communication or credit any overpayment to **Deposit Account No: 500482**. A duplicate copy of this authorization is enclosed.

__X__ A Check No. 4546 for the above specified full fee is enclosed. However, in case Applicant inadvertently miscalculated any required fee, the Commissioner is hereby authorized to charge the necessary additional amount associated with this communication or credit any overpayment to **Deposit Account No: 500482**. A duplicate copy of this authorization is enclosed.

This application is filed pursuant to 37 C.F.R. 1.53 in the name of the above-identified Inventor(s).

Please direct all correspondence concerning the above-identified application to the following address:

FERNANDEZ & ASSOCIATES, LLP
PATENT ATTORNEYS
PO BOX D
MENLO PARK, CA 94026-6204

(650) 325-4999
(650) 325-1203 : FAX
EMAIL: *iploft@iploft.com*

22877
PATENT_TRADEMARK OFFICE

Respectfully submitted,

DENNIS FERNANDEZ, ESQ.
Reg. No. 34,160

11/28/2000
Date

The Assistant Commissioner of Patents
United States Patent and Trademark Office
Washington, D.C. 20231
ATTN: Box Patent Application

Re: U.S. Utility Patent Application
Appl. No. (Not yet assigned); Filed 11/28/2000
For: **DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES**
Inventor(s): **NADJ, et al.**
Docket No.: **SIA-P032**

Sir:
The following documents are forwarded herewith for action by the U.S. Patent and Trademark Office:

1. U.S. UTILITY APPLICATION
entitled: **DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES**
having named inventor(s):
NADJ, et al.
a. a specification consisting:
(i) 27 pages prior to the claims, including title page;
(ii) 3 pages of claims;
(iii) 1 page abstract;
b. 13 sheets of **informal** drawings: (FIGs 1-13);
2. An unexecuted Combined Declaration and Power of Attorney by named inventors;
3. Form PTO-1082 (in duplicate);
4. An Assignment to **SILICON ACCESS NETWORKS, INC**.,
5. A return post card; and
6. Check No. 4546 for $ 395.00 to cover:

Patent application filing fee:	$ 355.00
Assignment Recordation fee:	$ 40.00
Excess claims fee:	$ .00

7. Verified Small Entity Status Statement with Cover Sheet

It is respectfully requested that the attached postcard be stamped with the filing date of the above documents and unofficial application number and returned to the addressee as soon as possible.

Respectfully submitted,

11/28/2000
Date

DENNIS FERNANDEZ, ESQ.
Reg. No. 34,160

FERNANDEZ & ASSOCIATES, LLP
PATENT ATTORNEYS
PO BOX D
MENLO PARK, CA 94026-6204
(650) 325-4999
(650) 325-1203 : FAX
EMAIL: *iploft@iploft.com*

UNITED STATES PATENT AND TRADEMARK OFFICE
DOCUMENT CLASSIFICATION BARCODE SHEET

Specification

4

Level – 2
Version 1.1

Application

For

United States Non-Provisional Utility Patent

Title:

DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES

Inventors:

Paul Nadj, residing at 1600 Scott Street, Tower B, Third Floor, Ottawa, Ontario, Canada, K1Y 4N7, a citizen of Canada.

David W. Carr, residing at 16 Sarrazin Way, Nepean, Ontario, Canada, K2J 3Z5, a citizen of Canada.

Edward D. Funnekotter, residing at 29 Delaware Avenue, #3, Ottawa, Ontario, Canada, K2P 0Z2, a citizen of Canada.

DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES

BACKGROUND INFORMATION

Field of the Invention

The present invention relates to generally to the field of sorting techniques and architectures.

Description of Related Art

Data structures known as heaps have been used previously to sort a set of values in ascending or descending order. Rather than storing the values in a fully sorted fashion, the values are "loosely" sorted such that the technique allows simple extraction of the lowest or greatest value from the structure. Exact sorting of the values in a heap is performed as the values are removed from the heap; i.e, the values are removed from the heap in sorted order. This makes a heap useful for sorting applications in which the values must be traversed in sorted order only once.

The properties of a heap data structure are as follows.

P1. A heap is a binary tree, or a k-ary tree where $k > 2$.

P2. A heap is a balanced tree; i.e., the depth of the tree for a set of values is bounded to $log_k(N)$, where N is the number of elements in the tree, and where k is described above.

P3. The values in a heap are stored such that a parent node is always of higher priority than all of its k descendent nodes. Higher priority means "higher priority to be removed from the heap".

P4. A heap is always left (or right) justified and only the bottom level may contain "holes" (a lack of values) on the right (or left) side of that level.

Property P2 is a reason that heaps are a popular method of sorting in systems where the sorted data must be traversed only once. The bounded depth provides a deterministic search time whereas a simple binary or *k*-ary tree structure does not.

Property P3 dictates that the root node of the tree always holds the highest priority value in the heap. In other words, it holds the next value to be removed from the heap since values are removed in sorted order. Therefore, repeatedly removing the root node removes the values in the heap in sorted order.

FIG. 1 is a conventional architectural diagram illustrating a tree-based heap data structure **10**, with a level 0 of heap, a level 1 of heap, a level 2 of heap, and a level 3 of heap. Tree-like data structures such as heaps are typically depicted and implemented as a series of nodes and pointers to nodes. Each node comprises a value to be sorted. In the level 0 of heap, a node **11** stores a value of 5. In the level 1 of heap, a node **12** stores a value of 22, and a node **13** stores a value of 10. In the level 2 of heap, a node **14** stores a value of 26, a node **15** stores a value of 23, a node **16** stores a value of 24, and a node **17** stores a value of 17. In the level 3 of heap, a node **18** stores a value of 27, and a node **19** stores a value of 38.

FIG. 2 is a conventional architectural diagram illustrating an array-based heap data structure **20**. It is well known in the art that balanced trees, such as heaps, may be constructed with arrays. The array-based heap data structure **20** eliminates the need to keep forward and backward pointers in the tree structure.

FIG. 3 is a conventional flow diagram illustrating the process of a heap remove operation **30**. Once a root node **11** is removed, a "hole" is created in the root node

position **11**. To fill the hole in the root node **11**, the bottom-most, right-most value (BRV) **12** is removed from the heap and is placed in the hole in the root node **11**. Then, the BRV and the *k* descendent nodes are examined and the highest priority value, if not the BRV itself, is swapped with the BRV. This continues down the heap. This comparison and swapping of values is known as the "percolate" operation.

FIG. 4 is a conventional flow diagram illustrating the process for a heap insert operation **40**. To add a value to be sorted into the heap, a slightly different kind of percolate operation is performed. The first hole **41** to the right of the bottom-most, right-most value is identified, and the new value is inserted there. This value is compared to the value in its parent node. If the new value is of higher priority than the parent value, the two values swap places. This continues until the new value is of lower priority, or until the root of the tree is reached. That is, the percolate continues *up* the tree structure rather than down it.

The described methods of adding and removing values to and from a heap inherently keeps a heap balanced: no additional data structures or algorithms are required to balance a heap. This means that heaps are as space-efficient as binary or *k-ary* trees even though the worst case operational performance of a heap is better than that of a simple tree.

A third operation is also possible: "swap". A swap operation consists of a remove operation whereby the BRV is not used to fill the resultant hole in the root node **11**. Instead, a new value is immediately re-inserted. The percolate operation is performed is identical to the delete case.

Because the percolate operations for remove and for insert traverse the data structure in different directions, parallelism and pipelining of the heap algorithm are inefficient and difficult, respectively.

High-speed implementations of heaps seek to find a way to execute the heap algorithm in hardware rather than in a software program. One such implementation is described in U.S. Pat. No. 5,603,023. This implementation uses a number of so-called "macrocells," each consisting of two storage elements. Each storage element can store one value residing in a heap. The two storage elements in a macrocell are connected to comparison logic such that the greater (or lesser) or the two can be determined and subsequently be output from the macrocell. A single so-called "comparing and rewriting control circuit" is connected to each macrocell so the comparisons between parent nodes and child nodes can be accommodated. In every case, both child nodes of a given parent are in the same macrocell, and the parent is in a different macrocell.

The shortcomings of the heap data structure and of previous implementations are described in the following points:

S1. Efficient pipelined heaps cannot be implemented due to opposing percolate operations.

There are two completely different percolate operations described in the previous section: one is used to remove values from the heap in sorted order, and one is used to insert new values into the heap. The former operation percolates downward from the top of the heap, whereas the latter operation percolates upward from the bottom of the heap.

A pipelined hardware operation is similar to an assembly line in a factory. In a pipelined heap – if such a structure existed – one insertion or removal operation would go through several stages to complete the operation, but another operation would be in the previous stage. Each operation goes through all the stages. I.e., if

stage S_j is currently processing operation i, stage S_{j-1} is currently processing operation $i+1$, stage S_{j-2} is currently processing operation $i+2$, and so on.

However, since some operations flow through the heap in one direction (e.g., insertion), whereas other operations flow though the heap in the other direction (e.g., removal), an efficient pipeline that supports a mix of the two operations is *difficult* to construct. This is because a removal operation needs to have current, accurate data in the root node (property P3, section 4.1) before it can begin, but an insertion of a new value percolates from the bottom up (see section 4.1). Thus, an insert operation is executed before a subsequent removal operation can be started. This is the direct opposite of a pipeline.

A unidirectional heap that operates only top-down is in the public domain. To operate in this fashion, the insert operation computes a path through the heap to the first unused value in the heap. Additionally, a simple method is proposed for tracking this first unused position. However, this tracking method assumes that heap property P4 holds. Although this property holds true for a traditional heap, removal of this property is desirable to eliminate shortcoming S2, described below. Thus, a suitable unidirectional heap structure suitable for high-speed pipelining does not exist in the current state of the art.

S2. Pipelined implementations of heaps are difficult to construct in high-speed applications due to the specifics of the "remove & percolate" operation.

The operation that removes values from a heap in sorted order leaves a "hole" in the root node once the highest priority value has been removed. This hole is filled with the bottom-most, right-most value in the heap.

In order to fill the hole caused by a remove operation, a hardware implementation of a heap must read the memory system associated with the current bottom of the tree to get the last value of the tree. This requires (a) that the location of the bottom always be known, and (b) that the all the RAM systems, except the tree root, run faster than otherwise necessary. When the each of the $log_k(N)$ tree levels of the heap has a dedicated RAM system, the required speedup is two times the speed otherwise required. (Placing the $log_k(N)$ tree levels of the heap in separate RAMs is the most efficient way to implement a pipelined heap, if such a thing existed, since it has the advantage of using the lowest speed RAMs for any given implementation.)

Point (b) states that "all" memory systems must be faster because the bottom of the heap can appear in any of the $log_k(N)$ memories.

Point (b) states that the memory must be twice as fast because the RAM is read first to get the value to fill the hole. The RAM may then be written to account for the

fact that the value has been removed. Later, if the downward percolation reaches the bottom level, the RAM will be again read and (potentially) written. Thus, a single operation may cause up to 4 accesses to RAM. Only 2 accesses are necessary if the remove operation is optimized to avoid reading and writing the bottom-most level to get the bottom-most, right-most value.

S3. A conventional design may not be fully pipelined. That is, since there is only one "comparing and rewriting control circuit," and since this circuit is required for every parent-child comparison in a percolate operation, it is difficult to have multiple parent-child comparisons from multiple heap-insert or heap-remove operations being processed simultaneously. This means that an insert or remove operation is executed before a new one is started.

S4. A conventional design is structured so that it takes longer to remove values from deeper heaps than from shallower heaps.

S5. A conventional design is incapable of automatically constructing a heap. An external central processor is repeatedly interacting with the design to build a sorted heap. (Once the heap is correctly constructed, however, the values may be removed in order without the intervention of the central processor).

S6. A conventional design employs so called "macrocells" that contain two special memory structures. Each macrocell is connected to a single so called "comparing and rewriting control circuit" that is required to perform the parent-child comparisons required for percolate operations.

This structure means that a macrocell is required for every pair of nodes in the heap, which in turn means that:

The structure does not efficiently scale to large heaps since large quantities of these special memory structures consume more area on a silicon die than would a traditional RAM memory sized to hold the same number of heap values.

The structure is costly to rework into a k-ary heap where $k>2$ since comparison logic grows more complex with the number of values being compared.

S7. A conventional design does nothing to prevent the painful problem of using a value from the bottom of the heap to fill the root node during a remove operation. The conventional design provides dedicated hardware to facilitate this nuance of heaps.

Accordingly, it is desirable to have a method and structure for a more efficient and flexible processing of a heap data structure.

SUMMARY OF THE INVENTION

The invention discloses a data structure known as a "pile", a method used to manipulate a pile, and a pipelined hardware implementation of a pile. Piles are designed using a heap or heap-like data structure. Heaps are data structures, and the algorithms that operate on these data structures, that are used for sorting values.

The benefits of piles are:

B1. In a pile, the percolate operations for both removal and insertion traverse the tree-like structure of the pile in the same direction, which allows a pipelined implementation to be constructed.

B1.1. It is widely understood that a pipelined implementation of a sorting algorithm, of which the pile is a class, performs more removal and insertion operations per unit of time than a non-pipelined implementation.

B1.2. It is obvious to one familiar with heaps that the time to get the result from a remove operation is independent of the depth of the heap. With a pipelined implementation of a heap another remove operation can begin once the first value is removed. This means that the latency of a remove operation is completely independent of the number of size of the heap.

B2. Piles do not require an implementation to access the bottom-most, right-most value in the heap before the percolate for a remove operation begins.

B2.1. This means that the effort expended in tracking the bottom-most, right-most value in a heap can be eliminated in the implementation of a pile.

B2.2. This allows the speed of the RAMs used to implement a pile to be cut in half. Slower RAMs are almost invariably cheaper than faster RAMs. Alternatively, if the speed of the RAMs is not reduced, it allows for a faster hardware implementation than is currently possible.

B4. An implementation of a pile can perform remove and insert operations without the intervention of a central processor.

B5. Conventional external RAM or conventional embedded RAM can be used to store the values in a pile. These RAMs are inexpensive compared to the custom memory cells used in the prior art.

B6. Multiple piles of dynamically changing sizes can exist in the same memory system since a pointer-based implementation of a pile consumes little additional memory.

Piles offer many advantages over heaps: they allow for fast, pipelined hardware implementations with increased throughput, making piles practical for a wide variety of new applications; piles remove the requirement to track and update the last position in the heap; piles reduce the number of memory reads accesses required during a delete operation; they require only ordinary, inexpensive RAM for storage in a fast, pipelined implementation; and they allow a random mixture of back-to-back insert, remove, and swap operations to be performed without stalling the pipeline.

The implementation of pile described herein has many advantages over the state of the art, and over what the advantages of piles vs. heaps otherwise implies: operations can be completed, for example, at fully ½ the speed of the RAM in a sustained fashion, and no intervention by a central processor is required to direct the heap algorithm.

Piles are used for implementing schedulers (e.g., priority queues, weighted fair queuing, traffic shaping). Piles offer a much more memory efficient solution than calendars, the typically implemented scheme for weighted fair queuing and traffic shaping. On high-speed links with many queues, efficient use of memory is required for a workable solution.

Heaps are often used in parallel processing systems to maintain a work queue. The processors consult the work queue when processors are idle so that processors may find additional work to do. Piles are a faster and improved heap that could be used in this

application. Dispatching work to processing units a network processor is one aspect of the parallel processing issue described above.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a conventional architectural diagram illustrating a tree-based heap data structure.

FIG. 2 is a conventional architectural diagram illustrating an array-based heap data structure.

FIG. 3 is a conventional flow diagram illustrating the process of a heap remove operation.

FIG. 4 is a conventional flow diagram illustrating the process for a heap insert operation.

FIG. 5 is an architectural diagram illustrating heaps that are constructed from miniature heaps in accordance with the present invention.

FIG. 6 is an architectural diagram illustrating a partitioning of a binary heap into supernodes with exceptions at the root in accordance with the present invention.

FIG. 7 is an architectural diagram illustrating a four-way heap that allows holes to percolate in any leaf node in accordance with the present invention.

FIG. 8 is an architectural diagram illustrating a four-way heap constructed with supernodes in accordance with the present invention.

FIG. 9 is an architectural diagram illustrating a pipelined heap implementation in accordance with the present invention.

FIG. 10 is an architectural diagram illustrating a pipeline resource table in accordance with the present invention.

FIG. 11 is an architectural diagram illustrating a multiple comparator blocks in a pipelined heap in accordance with the present invention.

FIG. 12 is an architectural diagram illustrating a pipelined heap with level caches in accordance with the present invention.

FIG. 13 is an architectural diagram illustrating a resource diagram showing use of level caches in accordance with the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENT(S)

Several aspects of piles are described below, which include heap remove operation, heap insert operation, combining an array implementation and a pointer implementation, a supernode structure, hole counters, multiple memory systems to construct a pipelined implementation of a heap-like data structure, multiple comparators to construct a pipelined heap implementation, and a pipelined heap with random commands, and a level cache to increase pipelined heaps processing.

1. Alteration of the heap remove operation, such that a hole may be percolated down the heap, with each hole behaving as the lowest priority value in the heap, and such that the hole may reside in any leaf position of the heap. The term leaf position applies equally well to an array-based implementation of a heap.

2. Alteration of the heap insert operation, such that the percolate operation operates on the heap data structure in a top-down rather than a bottom-up fashion, and such that the path followed by the percolate operation is not required to lead towards the first unused position in a traditional heap.

3. Using a combination of an array implementation and a pointer implementation of a heap to allow multiple dynamically-allocated pipelined heaps to co-exist within the same set of memories in an optimal fashion.

4. Combining nodes into a structure known as a "supernodes". A supernode is a set of k^2 sibling nodes from a *k-ary* tree, where $k >= 2$; and where each supernode requires only k pointers to the next tree level when a pointer implementation of a heap used.

5. Use of counters at each logical or physical pointer that count the number of holes that appear in the data structure referenced by the logical or physical pointer. These counters are known as "hole counters": hole counters guarantee a bounded-depth heap and they aid in dynamically resizing the heap.

6. A method that uses hole counters to aid in dynamically resizing the heap.

7. Use of multiple memory systems to construct a pipelined implementation of a heap-like data structure, where a memory system or a collection of memory systems represent a level or multiple levels of a heap-like data structure and where these memory systems may be accessed simultaneously.

8. The use of multiple comparators to construct a pipelined implementation of a heap-like data structure, where a comparator, or collection of comparators represent a level or multiple levels of a heap-like data structure and where these comparators may be actively doing work simultaneously.

9. Construction of a pipelined heap implementation capable of random mixture of insert, remove, and swap commands.

10. Use of a "level cache" to increase the speed of pipelined heaps beyond the point at which they would otherwise lose coherency.

Heap Remove Operation

A heap's remove operation requires that the last used position in a heap be constantly tracked so that the remove operation can find the last used position. The value in the last used position is used to replace the value removed from the root node.

This invention discloses a heap remove operation that entails allowing the hole itself, caused by removing the value in the root node, to percolate down the heap to any arbitrary leaf-node position. A hole is treated as the lowest priority value in the heap, with a priority equal to that of all other holes.

Since the heap does not grow in size when the removed value is replaced with a hole, the heap's overall depth remains bounded at a maximum of $log_k(N)$. However, the heap no longer satisfies property P4.

Since a hole is placed in the root node rather than a non-hole value from the bottom of the heap, there is no point in tracking the last used position of the heap.

Since a hole is considered to have the lowest priority in a heap, after the percolate operation is complete, a hole resulting from a delete operation will always reside in a leaf node of the tree.

Heap Insert Operation

A fast implementation of a heap is to have all the operations performed on the heap to access the levels of heap in the same order, either top-to-bottom or bottom-to-top. Note that the remove operation accesses the heap in top-to-bottom order. Rather than target only the bottom-most, left-most hole, the insert operation in the present invention may target *any* hole in the heap. This allows an insert operation to access levels of the heap in a top-to-bottom order.

Creating Multiple Heaps using an Array and Pointer Implementation

In a pipelined heap, it is advantageous to place different levels of the heap in different RAM systems. The fact that there are several RAMs rather than one does not impede an array-based implementation of a heap, as apparent to one skilled in the art.

An array-based implementation, however, has the disadvantage of being less flexible than a pointer based implementation since the various nodes may be easily rearranged in a pointer implementation simply by changing the pointers. An array-based implementation uses a fixed algorithm to determine parent and child nodes. This loss of flexibility makes it difficult to instantiate multiple heaps in the same memory system and further allow these instantiated heaps to grow and shrink in size (number of nodes) during the lifetime of the heaps.

A pointer-based implementation requires more memory than an array-based implementation since the pointer must be stored. A pointer-based implementation requires more time to traverse the heap than an array-based implementation since pointers may point to any given node in the memory system. This makes it difficult or impossible to guarantee that a long read, such as a DRAM burst, or such as is inherently possible with very wide bit memories, will read multiple nodes that are of immediate use to the heap algorithm.

To achieve the desirable properties of both array-based and pointer-based implementations in the same implementation, a combined structure may be used. FIG. 5 is an architectural diagram **50** illustrating heaps that are constructed from miniature heaps **51**, **52**, **53**, **54**, **55**, **56**, and **57**. This structure groups nodes together into miniature heaps and stores them in an array along with a series of pointers to the child (and possibly parent) miniature heap. The location of each array in memory may then be random.

This arrangement of the heap data introduces a new level scheme. Rather than counting logical levels of single nodes, levels of miniature heaps can be counted. Each of these levels can be placed in a single RAM system to allow parallel pipelined access.

Supernodes

A further refinement can be made to miniature heaps, which are shown in an architectural diagram **50** as shown in FIG. 5. The miniature heaps are just that: heaps. This means that when a value is inserted or removed, the nodes that exist within a miniature heap must be shuffled around to satisfy the heap property P3.

To avoid this shuffling of values, a new structure is used. Like the miniature heap structure, a group of nodes are co-located in memory such that the nodes may be read with a single long or wide read. However, the nodes that are grouped together out of the traditional heap are different than the previous case.

The nodes grouped together are k^2 sibling nodes from k parents. The exception to this is tree root, which may be k nodes; or the exception to this is the tree root and next level, which may be a single node and k nodes, respectively.

FIG. 6 is an architectural diagram **60** illustrating a partitioning of a binary heap ($k = 2$)_into supernodes with exceptions at a root node **61**. Nodes **62a** and **62b** forms a node group **62c**. Two node groups make up a supernode. Optionally, the node group **62c** can operate as the root of the heap. A supernode **63a** includes two node groups of **63b** and **63c**, where the node group **63b** contains nodes **63d** and **63e**, and the node group **63c** contains nodes **63f** and **63g**. In the bottom level of heap, two supernodes **64** and **65** are constructed.

The k^2 nodes in a supernode are arranged as k "node groups" each with k child nodes from a unique parent, and where each node group has an associated pointer to its child supernode. Note that the position of a node in a node group is related to the position of the node's child node group in a supernode.

This arrangement of nodes means three things: the potential of long and/or wide memory can be used since, for example, only one read must be performed to retrieve all the siblings of k nodes; heap percolate operations do not have to be performed within one of these blocks of memory; and fewer pointers are required than in the case of miniature heaps.

In summary, the idea behind supernodes is also that supernodes are a set of node groups placed in "adjacent" memory, such that either a wide read or a burst read will return the entire supernode. However, k-1 of the node groups in the supernode are not needed by the heap or pile operation (insert, remove, or swap) currently being executed: these k-1 node groups are for other paths down the heap that will not be traversed by the operation currently being executed. The supernode structure allows an operation to speculatively read data that it might need, before it knows exactly what it does need. This results in faster heap or pile operations because the required time to retrieve data from memory can pass in parallel with some of the heap algorithms. The data that the operation *does* need is typically ensured to be there but there is additional data that is not needed at that point in time. Thus, a supernode is not just an arbitary block of k^2 nodes. It is a block of k node groups, each with k nodes. The k node groups are siblings of each other in the heap, and only one sibling is needed for any given path through the heap. In

other words, supernodes are arranged in a data structure for speculatively reading children in a heap before the exact child is known.

This supernode structure is distinctly different from speculative reads in conventional heap implementations. In a conventional implementation the values that have been speculatively read are required to determine which values to keep. This means that the work of reading the data and the work of determine which data to keep cannot be done in parallel. With supernodes, the work can be done in parallel.

A *k-ary* heap (where *k=4*) that allows holes in any leaf position is shown in FIG. 7. FIG. 8 is an architectural diagram illustrating the same four-way heap, but constructed with supernodes. The supernode version of the heap is constructed using a node group for the root rather than a single node. This means that the maximum storage capacity of the heap with supernodes is one less than the other heap.

The remove operation for such a heap is as follows. This assumes that a k-way root node is used. Modification to derive the case for a single root node is obvious.

The root node group is read and the highest priority node is found and replaced with a hole. The value may be found by a k-way comparison. Since a node group has a pointer to its child supernode, the child supernode may be pre-fetched before the comparisons are started.

Once the comparisons are complete and the child supernode has been read from memory, (k-1) of the child node groups within the supernode may be discarded. The (k-1) child node groups were retrieved only to ensure that regardless of the result of the comparison on the root node, the correct child node would be available.

The remaining one node group of the supernode is examined to find the highest priority node. Also, since the node group has a pointer to its child supernode, the supernode may be pre-fetched before the comparison is started. The highest-priority value is exchanged with the hole in the parent node.

The remaining one node group is now treated as the root of a sub-heap, and the described steps repeat until the bottom of the heap is reached, or until the algorithm detects that a hole would be swapped with another hole.

The insert operation behaves similarly to the delete operation.

A different embodiment of the invention of supernodes entails keeping the values in a node group in sorted order to avoid comparisons during removal.

Use of Hole Counters at Each Logical or Physical Pointer

In a heap where holes are allowed, it becomes necessary to find these holes during an insert operation. An insert operation adds new values to a heap, and since a heap must abide by property P2 to give deterministic behavior, these values must occupy existing holes in the heap.

This invention describes a heap with holes that allows holes to occupy any leaf position. For an insert operation to ensure that a new value is swapped into a hole by the time percolation is complete, it needs to be able to find these "randomly" scattered holes.

In a pipelined implementation where each level of nodes (or miniature heaps, or supernodes) resides in a separate memory system, it is not productive to repeatedly read or write a level. Using a single bit at each pointer (or logical pointer in an array-based implementation) to indicate that there is a hole somewhere below in the heap does not

solve the problem since an operation does not know whether to change the state of the bit until it much later determines the number of holes that are present in the sub-heap.

Instead, a counter can be associated with every pointer. This counter is an accurate representation of the number of holes in the sub-heap below the pointer. Because any insert operation will ultimately succeed once it traverses a non-zero counter, each counter may be decremented as the pointer is traversed. There is no need to return to the counter later to update it.

Similarly, during a remove operation, it is guaranteed that a hole will be created under every pointer that is traversed. Therefore each counter may be incremented as each pointer is traversed.

Use of Multiple Memory Systems in a Heap for Pipelining

Pipelining allows a second operation to start before the first operation is finished, analogous to an assembly-line process.

Heaps are difficult or impossible to implement in a pipelined fashion in hardware because many memory accesses need to be performed on the same memory system. This contradicts the very definition of pipelining, which states that each unit of work to be done is performed by a dedicated resource, independent from all the other resources required to perform the previous or remaining work.

To pipeline a heap, nodes for each level of the heap are allocated from a different memory system. This allows one operation to be accessing one memory system whilst a subsequent operation is accessing another memory system.

However, the percolate operation swaps two values from two adjacent levels, so each stage in the pipeline requires access to two memory systems. The logic and RAM systems are laid out as shown in an architectural diagram **90** in FIG. 9

This arrangement allows an application to complete $log_k(N)$ more operations per second than previous implementations. For example, a 4-way pipelined pile realizes a five times speedup over a 4-way traditional heap when 1000 entries are sorted. Alternatively, this arrangement allows the memory to run at $1/(log_k(N))$ times the speed of a single memory system, *and maintain the same number of completed operations per unit time*. Memories that operate at lower speeds are typically cheaper than memories that operate at higher speeds.

The diagram and text show that each memory contains one level of a pipelined heap in a first level memory **93**, a second level memory **94**, and a third level memory **95**. Level A logic **91** reads and writes both the first level memory **93** and the second level memory **94**. Level B logic **92** reads and writes both the second level memory **94** and the third level memory **95**. Level A logic **91** can send information to level B logic **92** so that values can be percolated through the memories of the data structure in a top-to-bottom order. Note that a memory that operates at twice the speed as the other memories, for example, may support twice the number of heap levels. Such arrangements are included in this claim.

Because of inability to pipeline a heap, the only reason to place different tree levels of the heap in separate physical memories in a conventional design is to create a larger heap. However, placing the different tree levels of the heap in separate physical memories in a pipelined implementation is another feature in the present invention.

Furthermore, it should be noted that using several memory systems for the purposes of pipelining applies equally well to heaps constructed in other means, such as via miniature heaps and via supernodes. However, these examples are intended to be illustrative, and do not limit the scope of the present invention. An example pipeline resource diagram 100 is shown in FIG. 10. After heap request "A" (either an insert, remove, or swap operation) is read from Level 1 RAM in the first cycle, heap request "B" (either an insert, remove, or swap operation) is then pipelined and read during the seventh cycle. If a pipelined operation was not present, an arbitrary heap request "B" could not start until the eleventh cycle, the time when heap request "A" is fully complete. The time at which heap request "B" could start increases as the number of levels in the heap increases, and therefore increases as the number of values to sort increases.

Use of Multiple Comparator Blocks in a Heap for Pipelining

FIG. 11 is an architectural diagram **110** illustrating a multiple comparator blocks in a pipelined heap. Each memory system **114**, **115**, and **116** is associated with a respective block of comparators **111**, **112**, or **113**. Insert logic and comparator blocks **111**, **112**, and **113** include of k comparators that examine the k hole counters in a node group. Depending on the desired insert algorithm, the left-most hole counter, the right-most hole counter, or the largest hole counter, or the smallest hole counter could be

chosen as the winner of the *k* comparisons. The winning hole counter is used to direct the percolate operation down the heap.

The block further includes either *k* or one comparators that compare the value to be inserted with either the *k* node values or with the 1 winning node value. When *k* node values are compared, it should be understood that only the result of 1 comparison is kept: the result that corresponds to the winning hole counter. The winner of the value comparisons determines whether or not the new value to be inserted must be swapped with an existing value in the node group.

If the values are swapped, the new values is in the node group and the old value has been removed from the node group. The old value is given to the comparator block at the next level in the heap, and the procedure repeats.

The diagram shows "remove" comparator blocks **117** and **118**. These blocks each consist of *k* comparators that examine the *k* node values in a node group. The value with the highest priority is selected and removed from the node group. The value to be removed corresponds to a node group at the next level in the heap. The comparator block associated with that new level will fill the hole created at the original level with the winning value. This repeats down the heap.

Construction of a Pipelined Heap with Random Operations

There is no pipelined hardware implementation of a conventional heap that is capable of supporting a random mixture of insert, remove, and swap operations without stalling the pipeline to wait for an operation to complete. E.g., a heap that is not uni-directional, like the heap invented herein, needs to complete fully a series of insert operation before a remove operation can begin, although it may be possible to pipeline a series of like operations.

A pipelined heap implementation such as that shown in FIG. 11 is capable of a random mixture of any or all of insert & remove; insert & swap; remove & swap, and; insert, remove & swap.

Use of a Level Cache

The execution speed of a pipelined implementation of a heap that uses multiple comparator blocks and multiple memories is limited by the speed of the memories.

Behaviour of the Insert Operation

In this implementation, each *insert* request performs a memory read to retrieve a supernode. (At the root node and second tree level, only portions of supernodes need to be read). As previously described, a node group is isolated and comparisons are performed. A swap of the new value and a value in the node may be performed, altering the contents of the node group. The new node group must then be written back to memory. The memory system associated with the next level of the heap is then accessed, repeating the above operations.

This means that if the memory runs at X operations per second, $X/2$ insert operations per second can be completed.

Behaviour of the Remove Operation

In this implementation, each *remove* request performs a memory read to retrieve a supernode. A node group is isolated and comparisons are performed. A value to be removed is identified. At the root level, this value is returned to the initiator of the remove operation.

Removing the value leaves a hole. The altered node which now contains the hole need not be written back immediately. Recall that only a read has been performed so far. The next level supernode can be read, and the same steps are followed until a winning value is determined. This value is used to write back the original node group.

The sequence of events is in a four-level heap is as follows:

1. Read Level 1
2. Find Winner in Level 1
3. Read Level 2
4. Find Winner in Level 2
5. Write Level 1, using winner from Level 2
6. Read Level 3
7. Find Winner in Level 3
8. Write Level 2, using winner from Level 3
9. Read Level 4
10. Find Winner in Level 4
11. Write Level 3, using winner from Level 4
12. Write Level 4, containing a hole in place of the winner of Level 4

Thus, each memory system is accessed only twice, and a memory running at X operations per second is capable of $X/2$ heap remove operations per second.

Implications of the Above, and the Use of a Level Cache

Note that the time between reads and writes to the same memory, especially in the remove operation, is long. Comparisons need to be done to find the winner, and as memory speeds increase the time to perform these comparisons is significant. Because of this delay between the reads and writes, it is possible that an operation (operation 1) following another operation (operation 2) will read the same node group from memory that operation 1 is modifying, but has not yet written back in to the RAM. Operation 2, therefore, receives a stale copy of the data.

This problem may be solved either by reducing the rate of heap operations, or by increasing the speed of the memory. Either way, the theoretically maximum rate of *X/2* heap operations per second cannot be achieved. Another way to solve the problem is run the comparison operations faster. However, this can be expensive and technologically challenging when the speed required challenges the state of art for logic design and manufacturing.

One way to solve the problem is to implement a cache for node groups read from memory. When operation 2 accesses the same node group that operation 1 is modifying, operation 2 retrieves the data from the cache rather than from the memory. Because there is latency between the start of a read memory operation and the time at which the retrieved data is available, there is adequate time to consult the cache, and adequate time for operation 1 to complete its modifications to the cache. The *X/2* rate can be achieved with low comparison speeds even as the memory speeds increase.

The size of the cache is practical from an implementation standpoint. To allow any combination of requests that access the same nodes repeatedly, the cache depth only needs to have one entry per level. This is because requests need to be serviced sequentially in a heap to ensure correct data is available at all times, and therefore one request must finish modifications to a level before another request uses the data in that level.

This aspect of the invention also includes, however, different caching structures that contain more than one entry per level. This can be useful when statistical gain is exploited for higher performance. Recall that the cache is required when the node for one level is being modified but has not been committed to memory, and another request attempts to read that node from memory. If the length of time an implementation consumes to compute the "winners" for a level is long, the implementation can still use a high request rate and know (or hope) that the dynamics of the system are such that requests which are close in time will not typically access the same set of nodes. Accessing the same node "too soon" would force cause the request completion rate to temporarily slow down while the implementation waits for the nodes to have stable information.

In such a scheme many requests are being processed between a read from level *n* and a write to level *n*, many nodes must be cached. FIG. 12 is an architectural diagram **120** illustrating a pipelined heap with level caches, while FIG. 13 is an architectural diagram illustrating a resource diagram **130** showing use of level caches. A first level memory cache **121** is placed between a level A logic **123** and a level B logic **124**. A second level memory cache **122** is implemented between the level B logic **124** and a level C logic **125**. The first and second level memory caches 121 and 122 speedup the overall processing performance. On the other side, the level A logic 123 communicates information with a first level memory 126 and a second level memory 127, the level B logic 124 communicates with a second level memory 126 and a second level memory 127, and the level C logic 125 communicates with a first level memory 126 and a second level memory 127

The above embodiments are only illustrative of the principles of this invention and are not intended to limit the invention to the particular embodiments described. For example, one of ordinary skill in the art should recognize that the supernode concept can be selected as k node-groups, in which k denotes the number of optimal node-groups to suit a particular design. Accordingly, various modifications, adaptations, and combinations of various features of the described embodiments can be practiced without departing from the scope of the invention as set forth in the appended claims.

CLAIMS

WE CLAIM:

1. A data structure, comprising:

in a heap tree or similar data structure, comprising:

a root level having a node group, the node group having k number of nodes; and

a second level having one supernode, the supernode having k number of node groups.

2. The data structure of Claim 1, further comprising one or more holes in arbitrary leaf positions, the one or more holes representing absent values.

3. The data structure of Claim 1, wherein the k number of node groups are siblings of each other in the heap tree such that only one sibling node is needed for any given path in the heap tree.

4. The data structure of Claim 1, wherein the arrangement of the supernode in the heap tree allows for speculatively reading a children node in the heap tree before an exact desired child node is known.

5. The data structure of Claim 4, wherein the determination of the exact desired child proceeds in parallel with the retrieval of the supernode.

6. The data structure of Claim 1, further comprising a third level having k number of supernodes.

7. The data structure of Claim 2, further comprising a remove or delete operation which does not require a last value to be moved into a root node.

8. The data structure of Claim 7, wherein the remove or delete operation comprises:

removing the value from the root node; and

percolating the hole associated with the root node down the heap.

9. The data structure and remove operation of Claim 2, wherein the data structure contains a hole counter that counts the number of holes below the pointer for one or more of the pointers.

10. The data restructure of Claim 8, wherein the remove operation comprises incrementing by one the hole counter associated with each pointer that is traversed.

11. The data structure of Claim 2, further comprising an insert operation for percolating a value to be inserted starting at the root level and proceeding towards the bottom level.

12. The data structure of Claim 10, wherein an insert operation comprises:

percolating a value to be inserted starting at the root level;

in the one or more pointers, each pointer being associated with a hole counter that tracks the number of available holes, percolating the add value down a node in which the hole counter contains a value greater than zero; and

decrementing the selected hole counter by one.

5

ABSTRACT

An improved data structure is provided by modifying a public-domain data structure known as a "heap". When these improvements are applied, the resultant data structure is known as a "pile." This invention further described a pipelined hardware implementation of a pile. Piles offer many advantages over heaps: they allow for fast, pipelined hardware implementations with increased throughput, making piles practical for a wide variety of new applications; they remove the requirement to track and update the last position in the heap; they reduce the number of memory reads accesses required during a delete operation; they require only ordinary, inexpensive RAM for storage in a fast, pipelined implementation; and they allow a random mixture of back-to-back insert, remove, and swap operations to be performed without stalling the pipeline.

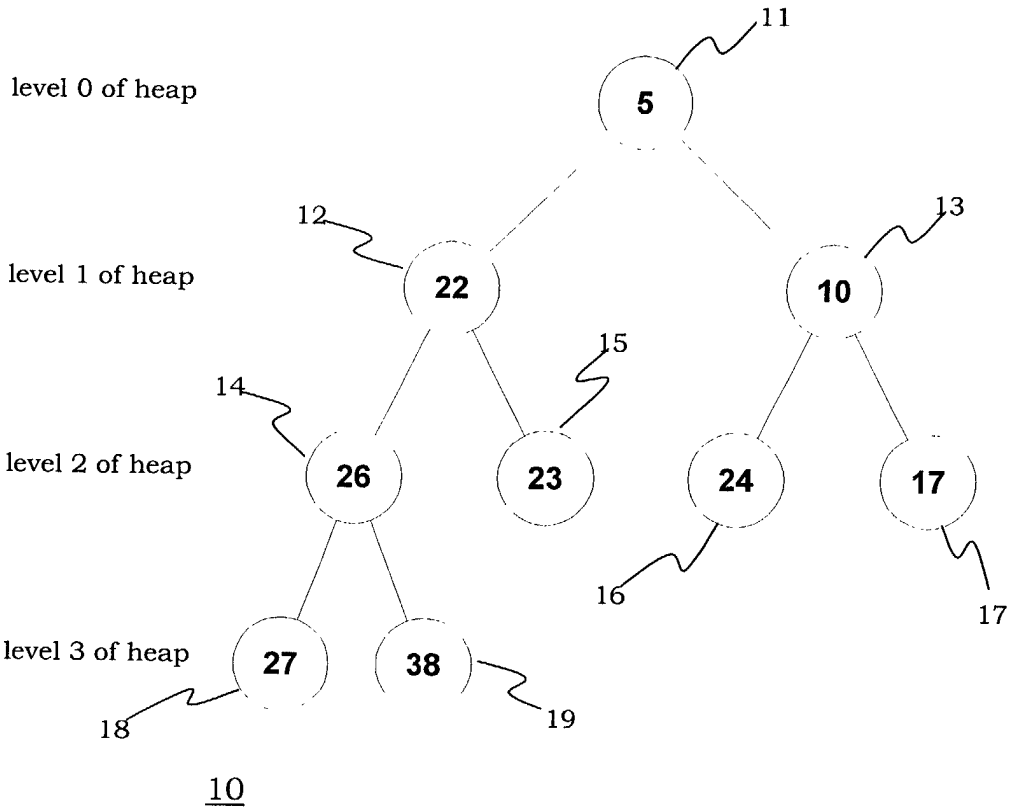


FIGURE 1
(Prior Art)

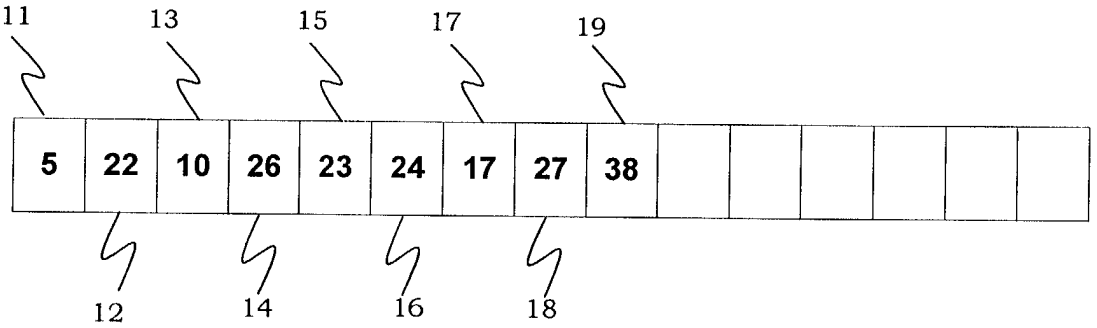


FIGURE 2
(Prior Art)

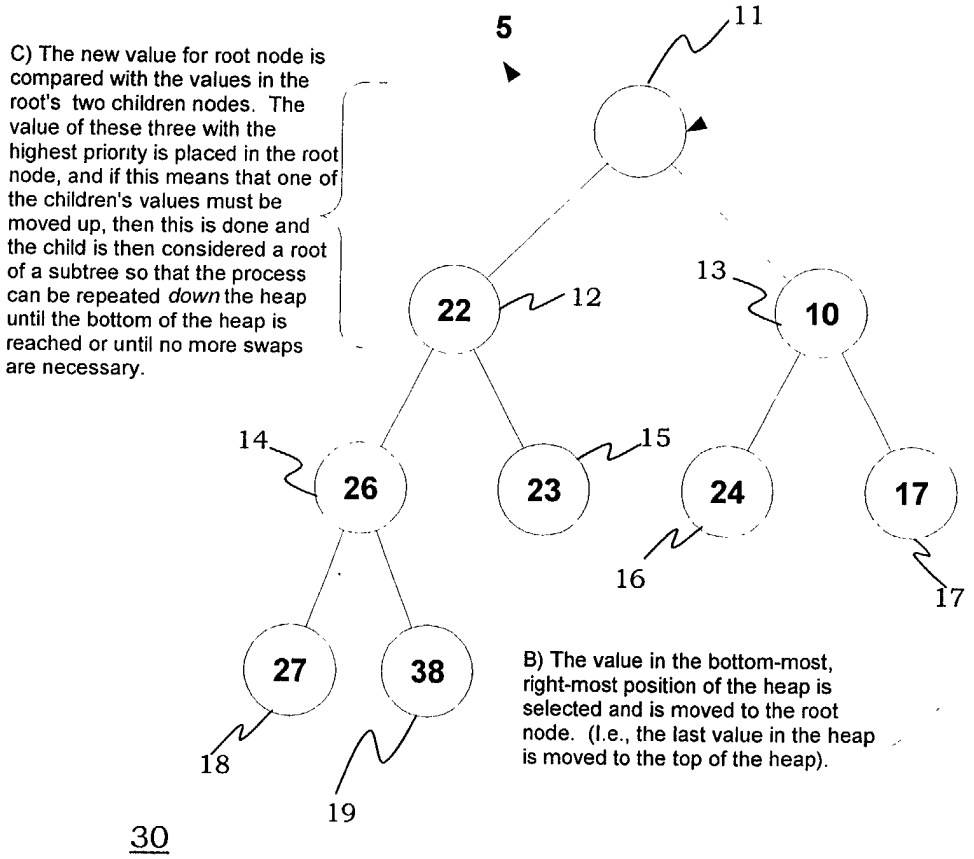


FIGURE 3

(Prior Art)

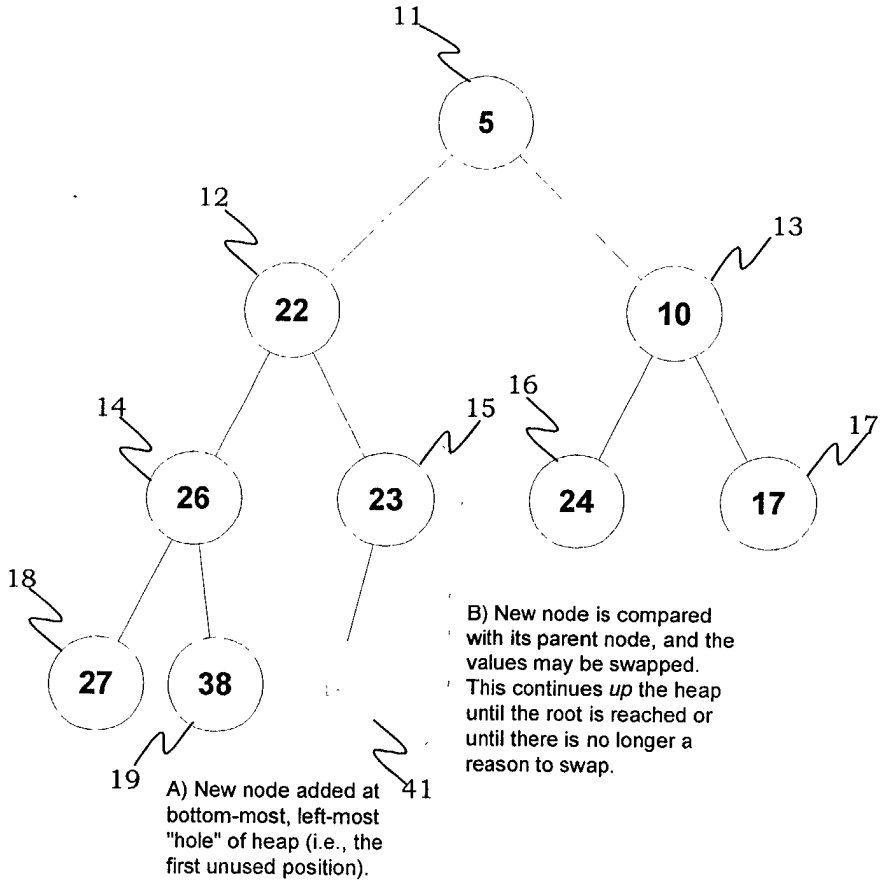
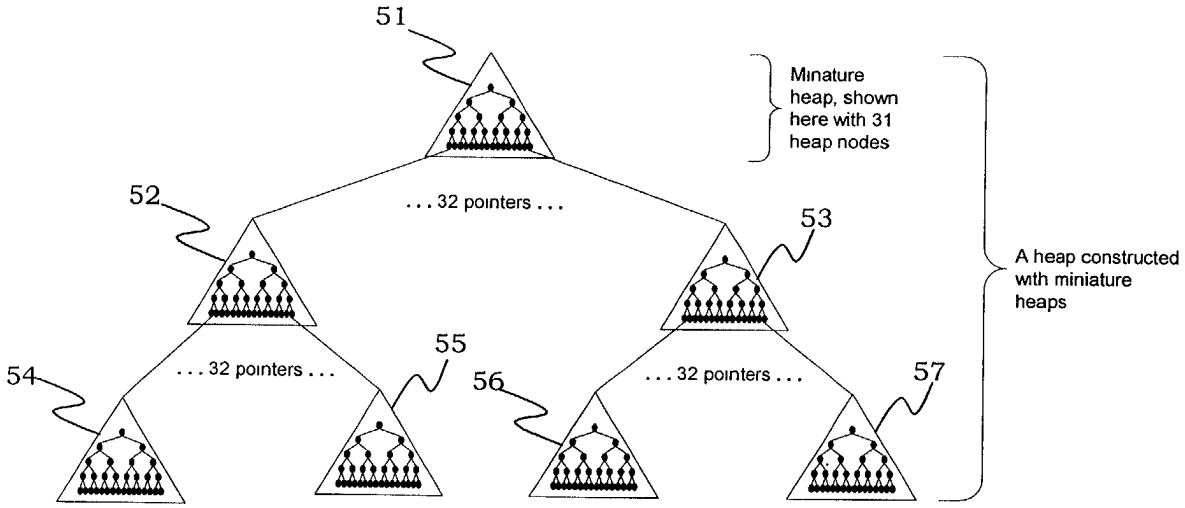
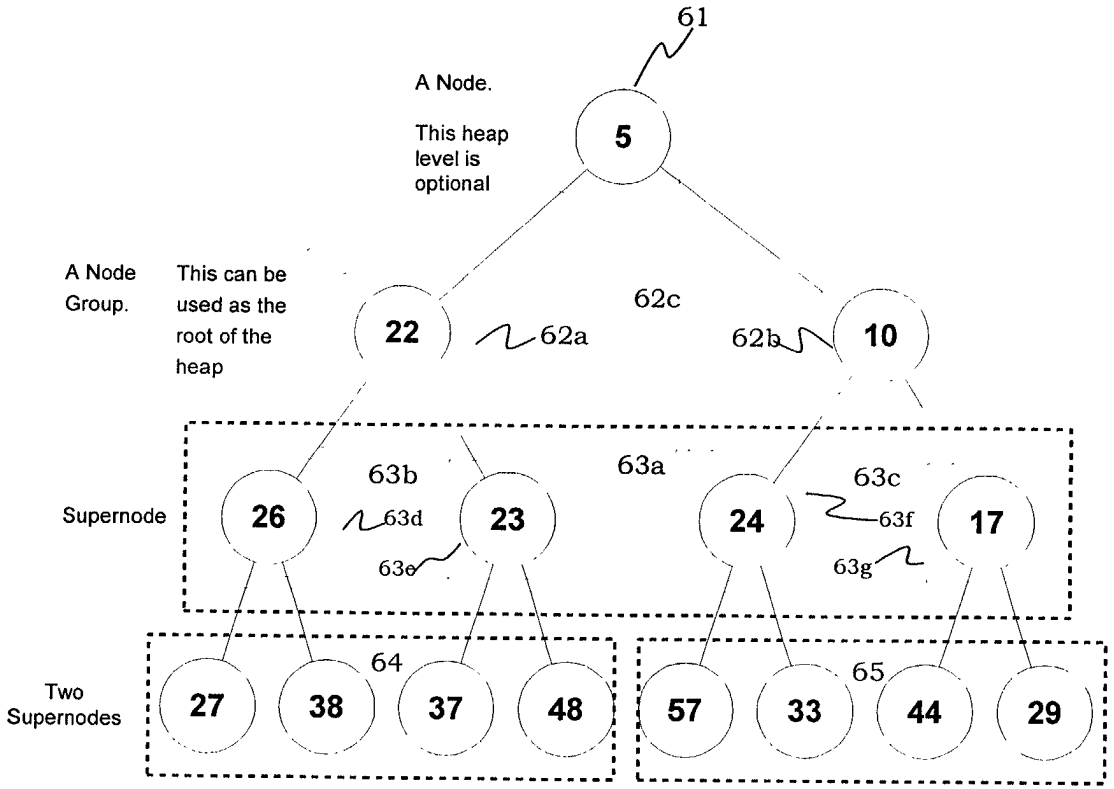


FIGURE 4
(Prior Art)


51
Minature heap, shown here with 31 heap nodes
52
. . . 32 pointers . . .
53
A heap constructed with miniature heaps
55
54
. . . 32 pointers . . .
56
. . . 32 pointers . . .
57

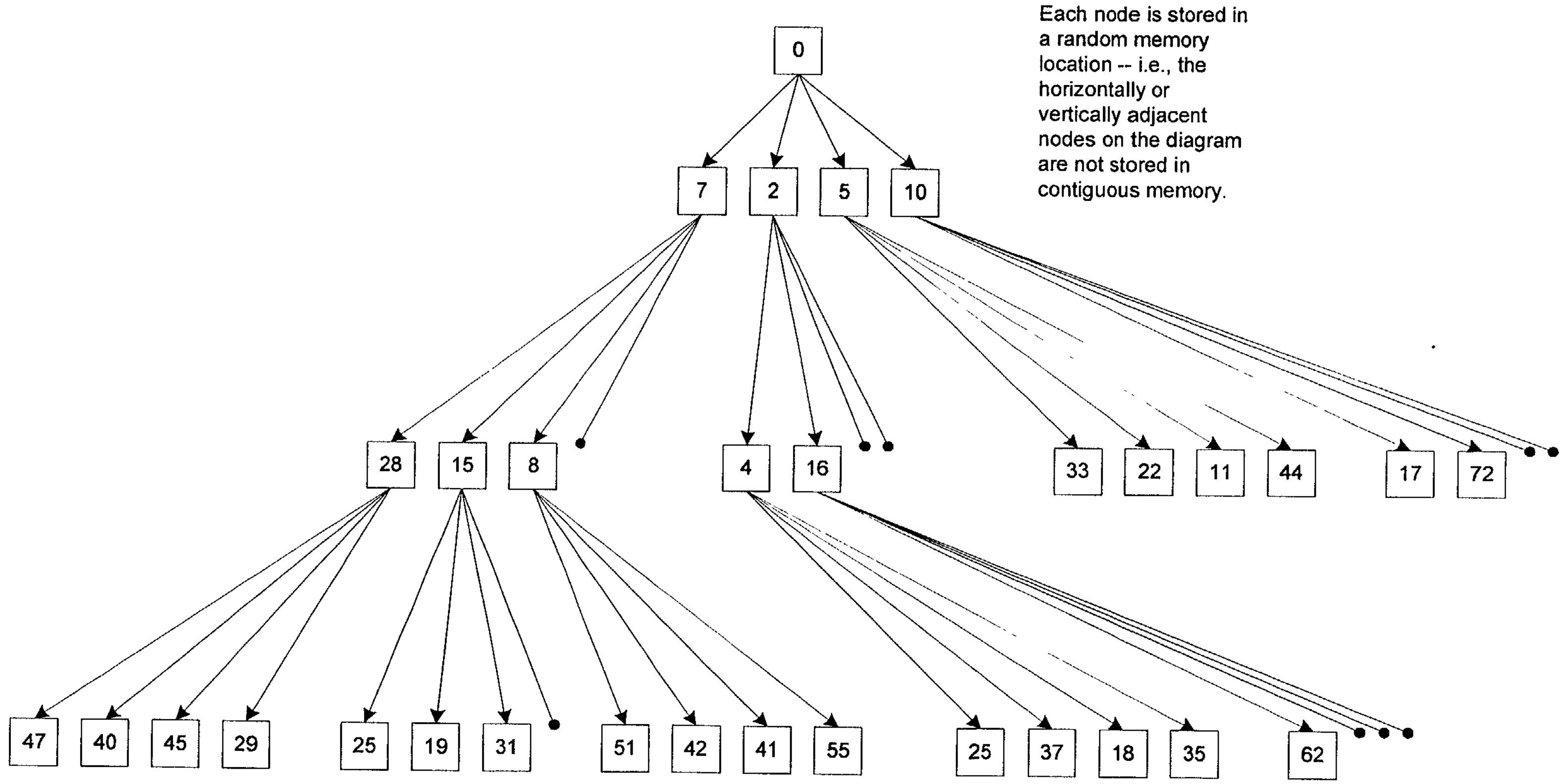
50

FIGURE 5



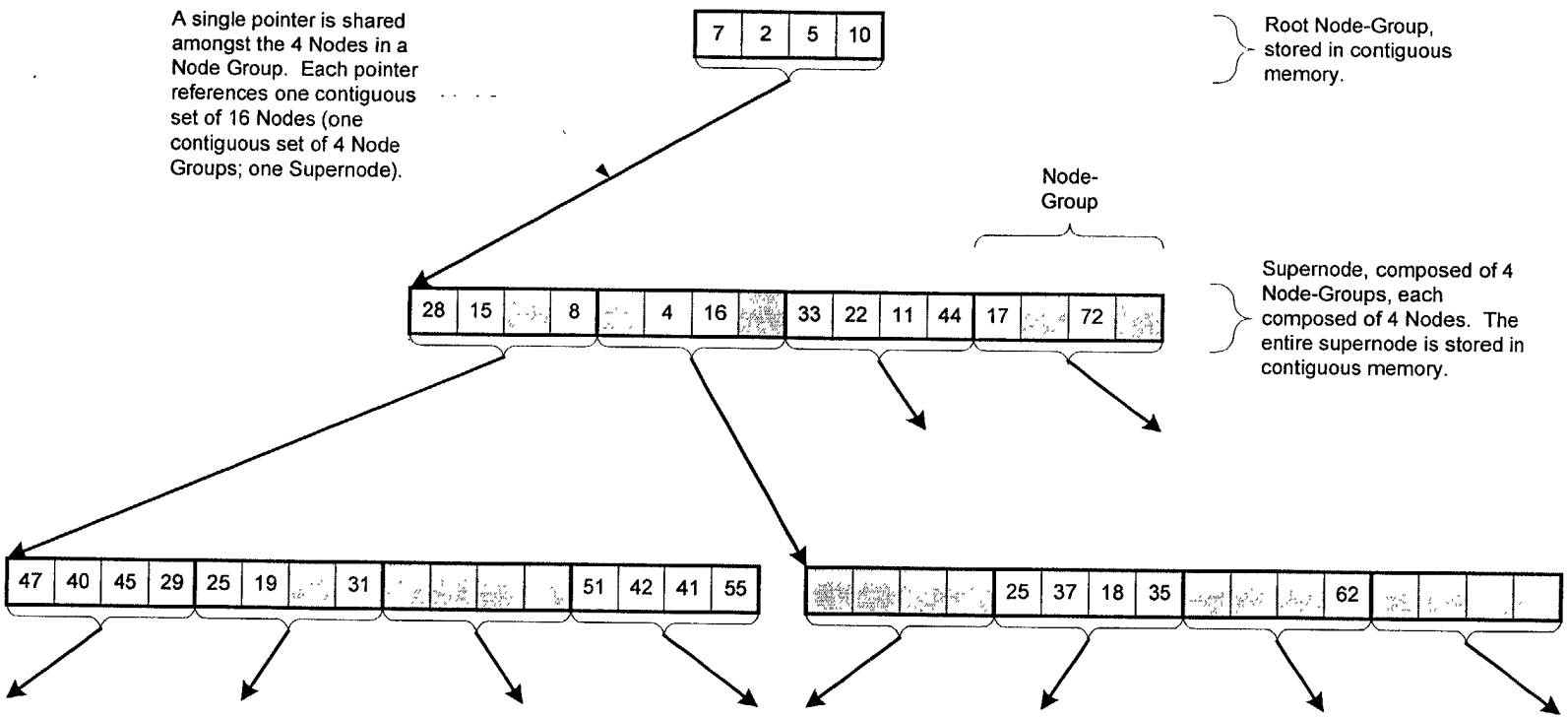
60

FIGURE 6


Each node is stored in a random memory location -- i.e., the horizontally or vertically adjacent nodes on the diagram are not stored in contiguous memory.
0
7
2
5
10
28
15
8
4
16
33
22
11
44
17
72
47
40
45
29
25
19
31
51
42
41
55
25
37
18
35
62

70

FIGURE 7


A single pointer is shared amongst the 4 Nodes in a Node Group. Each pointer references one contiguous set of 16 Nodes (one contiguous set of 4 Node Groups; one Supernode).
7 2 5 10
Root Node-Group, stored in contiguous memory.
Node-Group
28 15 8 4 16 33 22 11 44 17 72
Supernode, composed of 4 Node-Groups, each composed of 4 Nodes. The entire supernode is stored in contiguous memory.
47 40 45 29 25 19 31 51 42 41 55
25 37 18 35 62

80

FIGURE 8

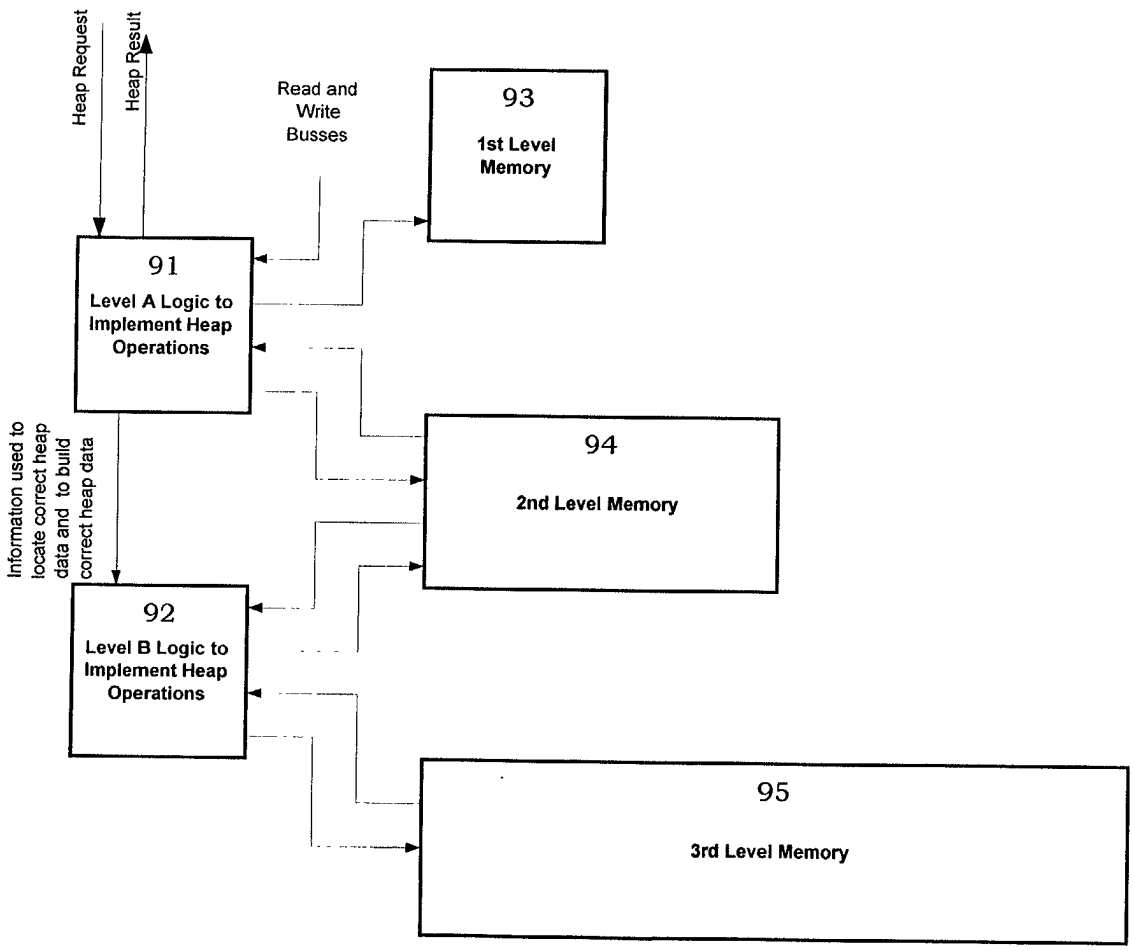
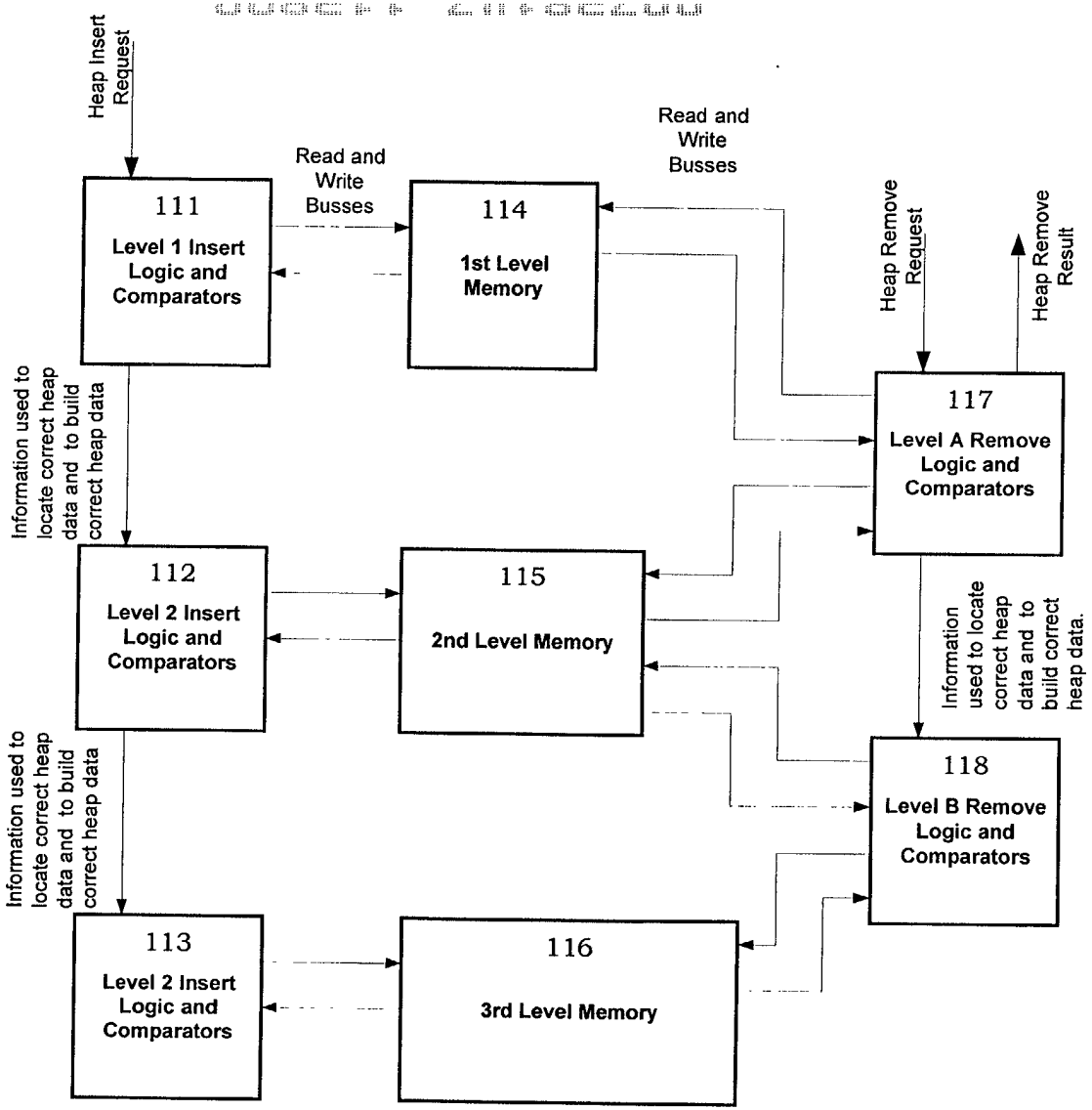

Heap Request
Heap Result
Read and Write Busses
93
1st Level Memory
91
Level A Logic to Implement Heap Operations
94
2nd Level Memory
Information used to locate correct heap data and to build correct heap data
92
Level B Logic to Implement Heap Operations
95
3rd Level Memory
90

FIGURE 9

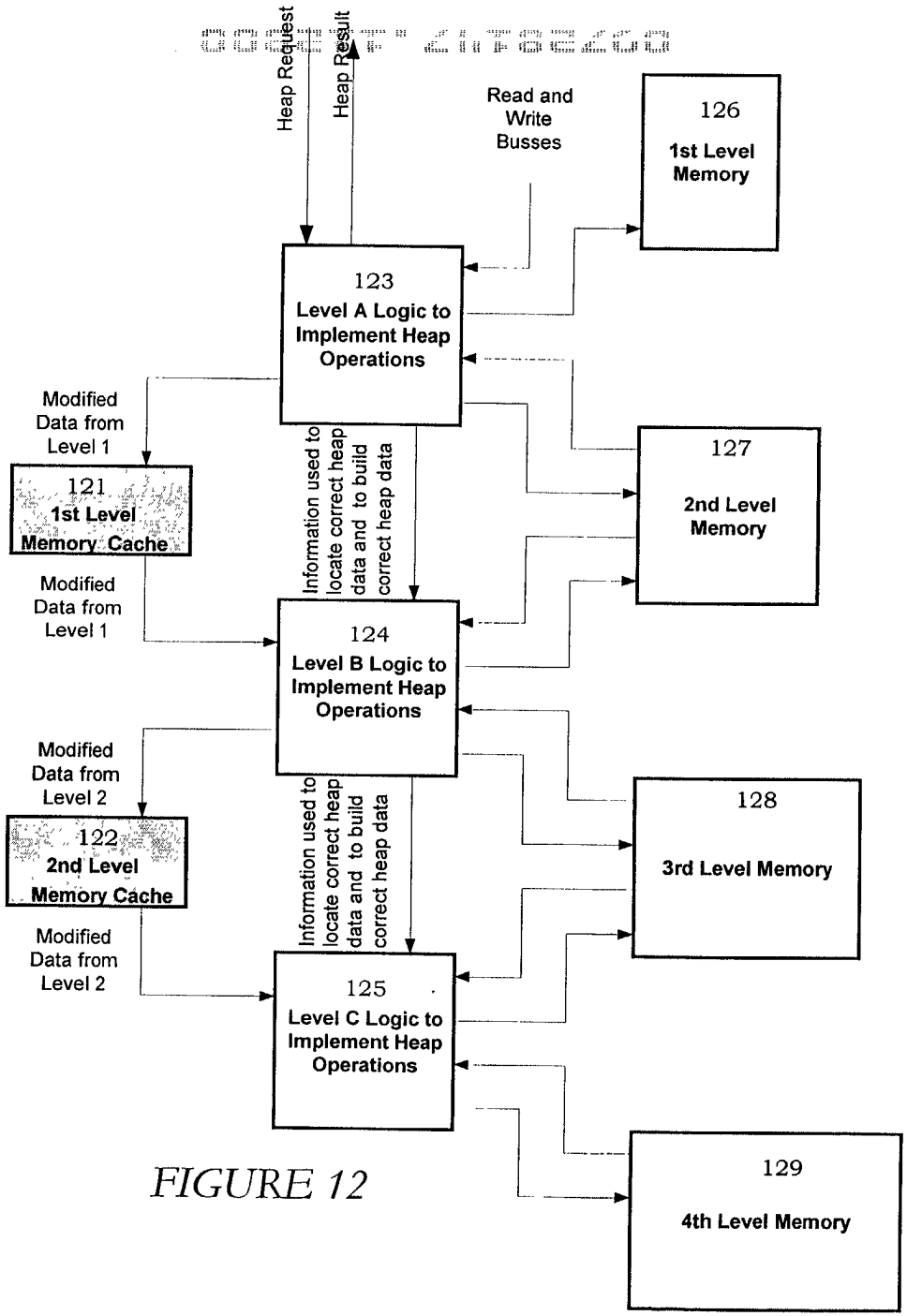
	time ---------->																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Read Level 1 RAM	A						B											
Write Level 1 RAM						A						B						
Level A Comparisons				A	A					B	B							
Read Level 2 RAM			A						B									
Write Level 2 RAM								A						B				
Level B Comparisons						A	A					B	B					
Read Level 3 RAM					A						B							
Write Level 3 RAM										A						B		
Level C Comparisons								A	A					B	B			
Read Level 4 RAM							A						B					
Write Level 4 RAM										A						B		

FIGURE 10



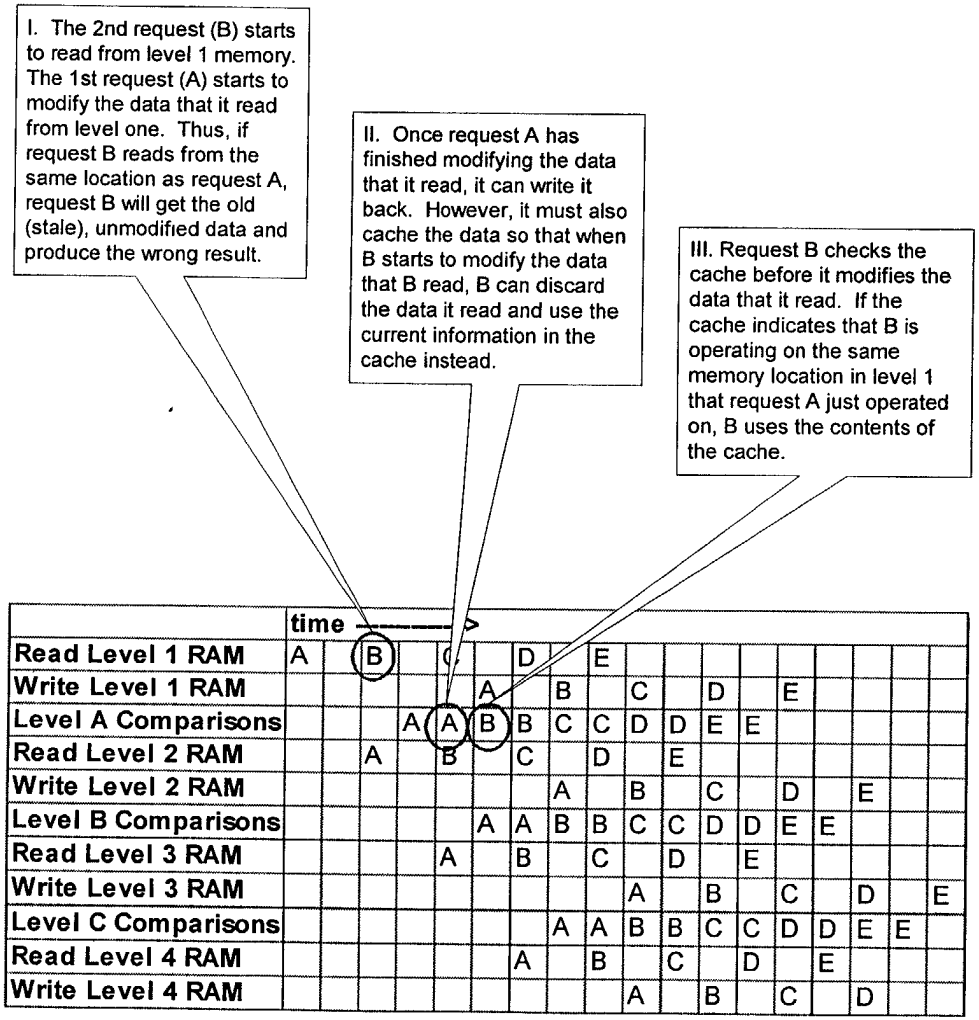
110

FIGURE 11



120

FIGURE 12



130

FIGURE 13

Docket No: **SIA-P032**

DECLARATION AND POWER OF ATTORNEY

As a below named inventor, I hereby declare that:

My residence, postal address and citizenship are as stated below next to my name.

I believe that I am the original, first and sole inventor (if only one name is listed below) or an original first and joint inventor (if multiple names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES

the specification of which is attached hereto unless the following information is indicated:
_____ was filed on _____;
as United States Application Number or PCT International Application Number _____;
and was amended on _____ (if applicable).

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendments referred to above.

I acknowledge the duty to disclose information that is material to patentability as defined in 37 CFR 1.56.

I hereby claim foreign priority benefits under 35 U.S.C. 119(a)-(d) or 365(b) of any foreign application(s) for patent or inventor's certificate, or 365(a) of any PCT International application which designated at least one country other than the United States, listed below and have also identified, as so indicated below, any foreign application for patent or inventor's certificate, or PCT International application having a filing date before that of the application on which priority is claimed.

Prior Foreign Application(s)			Priority Claimed
_____ (Application No.)	_____ (Country)	_____ (Day/Month/Year Filed)	___ Yes ___ No

I hereby claim the benefit under 35 U.S.C. 119(e) of any United States provisional application(s) listed below.

_____ (Application No.)	_____ (Day/Month/Year Filed)

I hereby claim the benefit under 35 U.S.C. 120 of any United States application(s), or 365(c) of any PCT International application designating the United States, listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States or PCT International application in the manner provided by the first paragraph of 35 U.S.C. 112, I acknowledge the duty to disclose information that is material to patentability as defined in 37 C.F.R. 1.56 that became available between the filing date of the prior application and the national or PCT International filing date of this application.

_____ (Application No.)	_____ (Day/Month/Year Filed)	_____ (Status - patented, pending, abandoned)

I hereby appoint the following attorney(s) and/or agent(s) to prosecute this application and to transact all business in the United States Patent and Trademark Office connected therewith:

DENNIS S. FERNANDEZ, REG. NO. *34,160*
PETER C. SU, REG. NO. *43,939*

Application No.: (Not yet assigned)

Send Correspondence and Communications to:

FERNANDEZ & ASSOCIATES, LLP
PATENT ATTORNEYS
PO BOX D
MENLO PARK, CA 94026-6204
(650) 325-4999
(650) 325-1203 : FAX
EMAIL: *iploft@iploft.com*

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and beliefs are believed to be true; and further that these statements were made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full name of sole or first inventor: **Paul Nadj**
Inventor's signature Date
Residence: **1600 Scott Street, Tower B, Third Floor, Ottawa, Ontario, Canada, K1Y 4N7**
Citizenship: **CANADA**
Postal Address: **SAME AS RESIDENCE**

Full name of sole or first inventor: **David W. Carr**
Inventor's signature Date
Residence: **16 Sarrazin Way, Nepean, Ontario, Canada, K2J 3Z5**
Citizenship: **CANADA**
Postal Address: **SAME AS RESIDENCE**

Full name of sole or first inventor: **Edward D. Funnekotter**
Inventor's signature Date
Residence: **29 Delaware Avenue, #3, Ottawa, Ontario, Canada, K2P 0Z2**
Citizenship: **CANADA**
Postal Address: **SAME AS RESIDENCE**

Docket No.: **SIA-P032**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of :
Application No.: (Not yet assigned) Group No.: (Not yet assigned)
Filed: 11/28/2000 Examiner: (Not yet assigned)

Title: **DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES**
Inventor(s): **NADJ, et al.**

Assistant Commissioner for Patents
Washington, D.C. 20231

SUBMISSION OF VERIFIED STATEMENT TO ESTABLISH SMALL ENTITY STATUS

The attached statement is being submitted to establish small entity status in this

_X_ Application
____ Patent

by the

____ Independent inventor(s) 37 CFR 1.9(c) and 1.27(b)
____ Non-inventor supporting claim by another 37 CFR 1.9(c) and 1.27(b)
_X_ Small Business Concern 37 CFR 1.9(d) and 1.27(c)
____ Nonprofit Organization 37 CFR 1.9(e) and 1.27(d)

Respectfully submitted,

DENNIS FERNANDEZ, ESQ.
Reg. No. 34,160

FERNANDEZ & ASSOCIATES, LLP
PATENT ATTORNEYS
PO BOX D
MENLO PARK, CA 94026-6204

(650) 325-4999
(650) 325-1203 : FAX
EMAIL: *iploft@iploft.com*

Docket No.: **SIA-P032**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Application No.: (Not yet assigned) Patent No.: (Not yet assigned)
Filed on: 11/28/2000 Issued on: (Not yet assigned)

Title: **DATA STRUCTURE AND METHOD FOR SORTING USING HEAP-SUPERNODES**
Inventor(s): **NADJ, et al.**

VERIFIED STATEMENT CLAIMING SMALL ENTITY STATUS

(37 CFR 1.9(f) and 1.27(c) – SMALL BUSINESS CONCERN)

I hereby declare that I am

____ the owner of the small business concern identified below:

__X__ an official of the small business concern empowered to act on behalf of the concern identified below:

Name of Small Concern:
SILICON ACCESS NETWORKS, INC.

Address of Small Concern:
211 RIVER OAKS PARKWAY
SAN JOSE, CA 95134

I hereby declare that the above identified small business concern qualifies as a small business concern, as provided in 37 CFR 1.9(d), for purpose of paying reduced fees to the United States Patent and Trademark Office, in that the number of employees of the concern, including those of its affiliates, does not exceed 500 persons.

I hereby declare that the rights under contract or law have been conveyed to, and remain with, the concern identified above with regard to the invention described in

__X__ the specification filed herewith, with title and inventor(s) as listed above.
____ the application identified above.
____ the patent identified above.

If the rights held by the above identified concern are not exclusive, each individual, concern, or organization having rights in the invention is listed below and no rights to the invention are held by any person, other than the inventor, who would not qualify as an independent inventor under 37 CFR 1.9(c), it that person made the invention, or by any concern which would not qualify as a small business concern under 37 CFR 1.9(d), or a nonprofit organization under 37 CFR 1.9(e).

Each person, concern or organization having any rights in the invention in addition to the above identified concern is listed below:
Name _____
Address _____
____ Individual __X__ Small Business ____ Nonprofit Organization

I acknowledge the duty to file, in this application or patent, notification of any change in status resulting in loss of entitlement to small entity status prior to paying, or at the time of paying, the earliest of the issue fee or any maintenance fee due after the date on which status as a small business entity is no longer appropriate. (37 CFR 1.28(b))

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true and further, that these statements were made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application, any patent issuing thereon, or any patent to which this verified statement is directed.

PERRY CONSTANTINE
Print Name of Person Signing

President and Chief Executive Officer and Board Member
Title of Person Signing

SILICON ACCESS NETWORKS, INC.
211 River Oaks Parkway, San Jose, CA 95134
Address of Person Signing

SIGNATURE _____ Date 11/27/50